JOURNAL COMPUTER SCIENCES INSTITUTE

JCSI 5 (2017) 213-219
Wysłane: 2017-06-07
Przyjęte: 2017-06-16

Analiza możliwości wykorzystania bibliotek Qt w wybranych zagadnieniach komunikacji międzyprocesowej

Łukasz Janusz Górniak*, Maciej Pańczyk
Politechnika Lubelska, Instytut Informatyki, Nadbystrzycka 36B, 20-618 Lublin, Polska

Streszczenie. W artykule scharakteryzowano czym jest Qt. Przedstawiono przykładową implementację związaną z komunikacją międzyprocesową wykorzystującą pamięć współdzieloną w klasycznym problemie producenta i konsumenta. Takie rozwiązanie jest mile widziane przez programistów, którzy z różnych powodów piszą swoje programy stosując framework Qt w możliwie szerokim zakresie.

Słowa kluczowe: komunikacja międzyprocesowa; C++; Qt; pamięć współdzielona

*Autor do korespondencji.

Adres/adresy e-mail: lukoog125@wp.pl

Analysis of Qt libraries usage in selected interprocess communication aplications

Łukasz Janusz Górniak*, Maciej Pańczyk
Institute of Computer Science, Lublin University of Technology, Nadbystrzycka 36B, 20-618 Lublin, Poland

Abstract. This article illustrates what Qt is. An example of the producer-consumer problem implementation in interprocess communication based on the shared memory is presented. Such solution can simplify the process of building interprocess application for those who want to write their programs using Qt in all possible aspects.

Keywords: inter-process communication; C++; Qt; shared memory

*Corresponding author.

E-mail address/addresses: lukoog125@wp.pl

1. Wstęp

Mechanizmy komunikacji międzyprocesowej najczęściej implementuje się z wykorzystaniem bibliotek systemowych [1]. Biblioteki systemowe są zazwyczaj wykonane w języku C [2]. Język C nie należy do najprostszych, zarówno jego nauka jak i programowanie w nim jest trudne i czasochłonne. W obecnych czasach zaś najdroższym elementem w procesie wytwarzania oprogramowania jest czas programisty, dlatego też powstają nowe języki wysokiego poziomu oraz różnego rodzaju biblioteki i frameworki, udostępniające wiele mechanizmów i komponentów wpływających znacznie na szybkość powstawania oprogramowania.

W ramach tego artykułu, przedstawiono alternatywę dla bibliotek systemowych, która ułatwi i przyspieszy proces implementacji komunikacji międzyprocesowej, a za razem pozwoli również na zbudowanie w pełni funkcjonalnego oprogramowania - z graficznym interfejsem użytkownika. Głównym celem jest analiza środowiska Qt oraz przedstawianie przykładowej implementacji komunikacji międzyprocesowej w celu oceny czy Qt spełnia wyżej zdefiniowane założenia. Qt umożliwia implementacje komunikacji międzyprocesowej kilkoma metodami - w artykule ocenie zostanie poddana metoda pamięci dzielonej [3].

2. Czym jest Qt

W tym rozdziale zostały przedstawione podstawowe informacje na temat środowiska Qt. Scharakteryzowano jego budowę, zastosowania oraz opisano specyficzne mechanizmy, które dobrze obrazują jakie ułatwienia zostały wprowadzone w stosunku do "czystego" języka C++. Pozwala to na określenie jakie zalety i wady niesie ze sobą korzystanie z Qt.

2.1. Historia i definicja Qt

Twórcami i pomysłodawcami Qt byli Haavard Nord i Eirik Chambe-Eng. Pierwsze prace nad biblioteką rozpoczęły się w 1991 roku. Aktualna wersja biblioteki to 5.8, wydana 23 stycznia 2017 roku [4].

Qt jest to zestaw bibliotek i narzędzi programistycznych przeznaczonych głównie dla języka C++ i QML (deklaratywnego język programowania opartego na JavaScript, służącego przede wszystkim do tworzenia graficznego interfejsu użytkownika), wspierających wieloplatformowość [5]. Pozwala na pisanie aplikacji desktopowych i mobilnych oraz dla systemów wbudowanych. W Qt można tworzyć proste aplikacje konsolowe, aplikacje z zaawansowanym graficznie interfejsem użytkownika i serwery. Qt może także zostać wykorzystane do budowania oprogramowania związanego z komunikacją międzyprocesową. Najpopularniejszymi wspieranymi

platformami są Linux, Windows, Mac OS X, Android i iOS [6].

Qt można zainstalować korzystając z instalatora w wersji online lub offline. Qt jest dostępne zarówno na licencjach wolnego oprogramowania jak i licencji komercyjnej [7].

2.2. Środowisko programistyczne

Jednym z głównych narzędzi, udostępnianych przez Qt jest zintegrowane środowisko programistyczne nazwane Qt Creator [8]. Qt Creator może być wykorzystany do nauki i tworzenia małych projektów, przez osoby rozpoczynające przygodę z Qt jak i do rozwijania dużych i zaawansowanych programów, przez doświadczonych programistów.

Tak jak w innych tego typu narzędziach, podstawowymi funkcjonalnościami są: możliwość utworzenia i skonfigurowania projektu, edycji plików oraz łatwa praca z kompilatorami. Qt Creator dostarcza rozbudowany edytor kodu który dba o poprawną składnie kodu, kolorowanie kodu, podkreślanie błędów, generowanie kodu, formatowanie kodu czy wyszukiwanie danej frazy w kodzie. W przypadku aplikacji z graficznym interfejsem użytkownika, można skorzystać z wbudowanych edytorów, pozwalających na budowę interfejsu metodą "przeciągnij i upuść". Qt dostarcza dwa tego typu edytory w zależności czy interfejs graficzny jest budowany w QML (nowoczesny wygląd) czy z wykorzystaniem Qt Widgets (tradycyjny wygląd). Z poziomu edytora można zdefiniować powiązania pomiędzy obiektami, począwszy od układu, po oddziaływanie na siebie (mechanizm sygnałów i slotów), jak i określić ich właściwości np. nazwę, wielkość, kolor, marginesy, czcionki i wiele innych [8].

Qt Creator współpracuje również z różnymi zewnętrznymi narzędziami [9]. Jednym z takich narzędzi jest debuger. Środowisko wspiera wiele różnych debugerów przy pomocy których można łatwo śledzić prace aplikacji, wyniki są prezentowane w postaci wizualnej. Kolejnymi wspieranymi narzędziami są systemy kontroli wersji, narzędzia do porównywania kodu oraz symulatory urządzeń np. do zastosowań w przypadku rozwijania aplikacji mobilnych.

2.3. Mechanizmy w Qt

C++ to obiektowy język programowania, zaprojektowany przez Bjarne Stroustrupa, jako rozszerzenie języka C. Charakteryzuje się wysoką wydajnością, jest językiem ogólnego przeznaczenia. Programowanie aplikacji w tym języku zazwyczaj wymaga korzystania z niestandardowych bibliotek. Qt można nazwać pewnego rodzaju nakładką na C++, która ma na celu ułatwienie i przyspieszenie wytwarzania oprogramowania. Aby osiągnąć taki rezultat, stosuje się różnego rodzaju kombinacje makr, sztuczki z preprocesorem i dodatkowe narzędzia, a więc stylu programowania używanego w aplikacjach Qt nie można określić mianem "czystego" C++. Pomimo wprowadzonych usprawnień kod napisany w Qt może być swobodnie łączony z kodem w C/C++.

Qt dostarcza kilka specyficznych mechanizmów, które zwiększają możliwości bazowego języka. Implementacja większości z tych mechanizmów, opiera sie na specjalnej klasie o nazwie QObject. W tym rozdziale opisano trzy podstawowe mechanizmy.

2.3.1. Meta-objekty (Meta-Object System)

Metaobiektem określa się obiekt, który opisuje strukturę innego obiektu. System meta-objektów zaimplementowany w Qt to mechanizm który pozwala na:

- uzyskiwanie informacji o faktycznym typie obiektu, podczas wykonywania programu - RTTI (Run Time Type Information), nawet jeśli programista dysponuje tylko wskaźnikiem na klasę bazową,
- komunikację miedzy obiektami - mechanizm sygnałów i slotów,
- stanowi bazę dla dynamicznego systemu właściwości obiektu (Property System).

Na mechanizm meta-objektów składają się trzy podstawowe elementy [10]:

- QObject - aby korzystać z mechanizmu meta-obiektów, należy dziedziczyć z tej klasy. Do uzyskania podstawowych informacji o obiekcie wykorzystuje się metodę QObject::metaObject() zwracającą obiekt klasy QMetaObject, która udostępnia szerokie API, pozwalające na pozyskanie informacji o danej klasie np. nazwę klasy, ilość oraz szczegóły zdefiniowanych metod, ilość dynamicznych i statycznych właściwości zawartych w klasie oraz szczegóły o każdym z tych elementów, informacje o typach wyliczeniowych i wiele innych. Każdy obiekt może także posiadać własną nazwę - metody setObjectName() i objectName(). Qt udostępnia, do identyfikacji typów, oprócz standardowych operatorów z języka C++ (dynamic_cast oraz typeid) dodatkowy operator qobject_cast oraz metodę QOBject::inherits(), która przyjmuje jako parametr nazwę klasy,
- Q_OBJECT - dziedziczenie z QObject nie jest jedynym wymaganiem jakie musi być zrealizowane w celu korzystania z systemu meta-obiektów. Kolejnym wymaganiem jest umieszczenie specjalnego makra w sekcji prywatnej danej klasy. Makro Q_OBJECT definiuje dodatkowe metody i pola, które pozwalają na korzystanie z mechanizmu meta-obiektów oraz z mechanizmu sygnałów i slotów,
- Meta-Object Compiler (moc) - jest to specjalny kompilator, którego zadaniem jest generowanie plików implementacyjnych dla klas, które zawierają makro Q_OBJECT. Pliki zawierają implementacje mechanizmu meta-objektów dla tych klas.

2.3.2. Sygnały i sloty (Signals and Slots)

Mechanizm sygnałów i slotów bazuje na systemie meta-objektów [11]. Jest to rozszerzenie pozwalające na łatwą komunikacje pomiędzy obiektami. Ten mechanizm to implementacja wzorca projektowego o nazwie obserwator (inaczej nazywanego Wydawca-Prenumerator) [12].

Sygnały i sloty są wykorzystywane zarówno w aplikacjach z graficznym interfejsem użytkownika jak i w programach konsolowych. Przykładowo sygnał może odpowiadać zdarzeniu kliknięcia na przycisk, a slot reakcji na to zdarzenie. Do powiązania sygnału ze slotem wykorzystuje sie metodę QObject::connect(). Sygnały można łączyć zarówno z slotami jak i z innymi sygnałami. Mechanizm ten wymaga, aby w aplikacji była uruchomiona pętla zdarzeń (mechanizm przetwarzający zdarzenia) [8].

2.3.3. Relacje pomiędzy obiektami (Object Trees)

Kolejnym mechanizmem jaki dostarcza Qt jest możliwość układania obiektów w strukturę drzewa. Jest to implementacja wzorca projektowego o nazwie kompozyt [12]. Taka relacja (rodzic-dziecko) pozwala na łatwiejsze zarządzanie obiektami.

Klasy zbudowane w oparciu o system meta-objektów udostępniają metodę QObject::setParent() oraz QObject::parent(), które pozwalają na ustawienie rodzica dla danego obiektu oraz pobranie rodzica tego obiektu. Kolejną udostępnianą metoda jest metoda children(), która zwraca wszystkie dzieci danego obiektu.

Założenie jest takie, że jeden obiekt może mieć jednego rodzica i wiele dzieci. Takie podejście pozwala na łatwe przeglądanie struktury obiektów, jakie zawiera dany obiekt (rodzic), co pozwala na automatyzacje procesu zwalniania pamięci - Qt podchodzi do tego w ten sposób, że w momencie niszczenia obiektu rodzica, są automatycznie niszczone obiekty dzieci [13].

3. Przykłady komunikacji międzyprocesowej z zastosowaniem Qt

Qt jest zbudowane z wielu modułów i ma szeroki zakres zastosowań, umożliwia tworzenie rozbudowanych graficznie aplikacji oraz zaawansowanych programów konsolowych. Dostarcza również kilka dróg pozwalających na implementacje komunikacji międzyprocesowej. W tym przykładzie skupiono się na metodzie pamięci dzielonej.

Jednym z problemów, jakie spotyka się implementując przesyłanie danych pomiędzy współpracującymi ze sobą współbieżnymi procesami jest "problem producenta i konsumenta". Problem producenta i konsumenta należy do grupy klasycznych informatycznych problemów synchronizacji. Podstawowe założenia tego problemu wyglądają następująco:

- jeden proces (producent) jest odpowiedzialny za generowanie (produkcje) danych, natomiast zadaniem drugiego (konsument) jest odbiór danych (produktów),
- wytworzone produkty są umieszczane w magazynie,
- nie można umieszczać produktów w pełnym magazynie,
- nie można pobierać produktów z pustego magazynu.

W ramach analizy wykonano opis klasy wykorzystanej w przykładzie, dokonano przeglądu interfejsu tej klasy oraz przedstawiono przykładową implementacje rozwiązującą wyżej zdefiniowany problem.

3.1. Opis dostarczonych klas i ich interfejsów

Jednym z najszybszych sposobów komunikacji międzyprocesowej jest pamięć współdzielona. Qt umożliwia użycie tej metody komunikacji wykorzystując klasę QSharedMemory [14]. Klasa QSharedMemory została zaprojektowana w celu stworzenia prostego interfejsu pozwalającego na implementację wymiany danych poprzez segment pamięci współdzielonej. Dodatkowym usprawnieniem jest wbudowany mechanizm zapewniający spójność danych, wykorzystujący semafory [1].

W tabeli 1, został przedstawiony opis najważniejszych metod jakie są dostępne do dyspozycji programisty. Klasa udostępnia również dwa typy wyliczeniowe: AccessMode oraz SharedMemoryError. Typ AccessMode opisuje tryby dostępu do pamięci. Dostęp do pamięci może być realizowany na dwa sposoby: ReadWrite oraz ReadOnly. Domyślnym trybem jest ReadWrite, który daje możliwość zapisu danych do pamięci jak i odczyt z pamięci. Natomiast typ wyliczeniowy o nazwie SharedMemoryError dostarcza listę błędów, które mogą wystąpić podczas wykonywania programu.

3.2. Badania

W ramach badań przygotowano dwie aplikacje, które rozwiązują problem producenta i konsumenta. Założono, że producent ma za zadanie wyprodukowanie 200000 produktów i dysponuje magazynem o ograniczonej wielkości 4096 bajtów.

Aplikacje składają się z dwóch głównych klas: Producent oraz Konsument. Obie klasy implementują metodę obsluzBlad(), jest to metoda o bardzo prostej logice - w przypadku wystąpienia błędu wysyła komunikat na wyjście i kończy działanie aplikacji. Metoda przyjmuje dwa argumenty, pierwszy określający czy wystąpił błąd, drugi zawierający treść komunikatu, jaki ma być wyświetlony w przypadku błędu.

Listing 1 przedstawia interfejs klasy Producent. W klasie została zadeklarowana metoda start(), która rozpoczyna "produkcje" oraz metoda losujLiczbe(), pozwalająca na generowanie pseudolosowych liczb z zadanego zakresu (produktów), a także wspomniana wyżej metoda obsluzBlad(). W sekcji prywatnej umieszczono deklaracje typów: sc_klucz jest to zmienna przechowująca klucz do pamięci współdzielonej w postaci łańcucha znaków, sc_maxIloscProduktow i sc_maxRozmiarPamieci określają kolejno ilość produktów do wyprodukowania oraz maksymalny rozmiar danych jakie mogą zostać umieszczone w pamięci, m_pamiecDzielona to wskaźnik na obiekt wystawiający interfejs do operowania na segmentach pamięci współdzielonej, zmienna m_wyprodukowano jest wykorzystywana do zliczania wyprodukowanych produktów, natomiast obiekt m_dane gromadzi wyprodukowane produkty (liczby) w postaci tablicy bajtów.

Tabela 1. Opis metod klasy QSharedMemory [14].

Typ zwracany	Nazwa metody	Opis
	QSharedMemory(const QString &key, QObject *parent = Q_NULLPTR)	Konstruuje obiekt o zadanym kluczu i rodzicu. Aby utworzyć segment pamięci współdzielonej należy wywołać metodę create(). W celu dołączenia do już istniejącego segmentu, należy wywołać metodę attach().
	~QSharedMemory()	W destruktorze następuje usunięcie klucza oraz odłączenie od pamięci współdzielonej. Jeśli to był ostatni z obiektów połączonych z pamięcią współdzieloną, następuje usunięcie segmentu pamięci współdzielonej.
bool	attach(AccessMode mode = ReadWrite)	Metoda podejmuje próbę dowiązania do segmentu pamięci współdzielonej. Dostęp do pamięci może być realizowany w różnych trybach - domyślnym trybem jest ReadWrite. Gdy operacja się powiedzie, jest zwracane wartość true, w przypadku niepowodzenia wartość false. Metoda error() może zostać wykorzystana do sprawdzenia błędów jakie wystąpiły.
Bool	create(int size, AccessMode mode = ReadWrite)	Wykorzystywana do utworzenia segmentu pamięci współdzielonej. Pierwszy przekazywany parametr określa wielkość segmentu pamięci w bajtach, drugi tryb dostępu. Segment pamięci jest tworzony w oparciu o klucz przekazany w konstruktorze klasy lub ustawiony za pomocą metod setKey() lub setNativeKey(). Jeżeli istnieje pamięć współdzielona o zadanym kluczy, jest zwracane false. Metoda error() może zostać wykorzystana do sprawdzenia błędów jakie wystąpiły.
void *	data()	Metoda zwraca wskaźnik na zawartość pamięci współdzielonej. W przypadku gdy segment pamięci nie istnieje jest zwracany null. Przed operacjami zapisu lub odczytu z pamięci, należy wykorzystać metody lock() lub unlock() w celu synchronizacji dostępu.
Bool	detach()	Odłącza obiekt od pamięci współdzielonej, zwraca true w przypadku powodzenia lub false jeśli operacja się nie powiodła lub jeśli obiekt nie był dowiązany. Wartość false może być także zwrócona w przypadku gdy nie nastąpiło odłączenie ponieważ segment pamięci był zablokowany przez inny proces.
SharedMemoryError	error() const	Zwraca numer błędu opisanego za pomocą enum. W przypadku niewystąpienia błędu jest zwracana wartość QSharedMemory::NoError.
QString	errorString() const	Zwraca opis błędu w postaci łańcucha znaków.
QString	key() const	Zwraca klucz do pamięci współdzielonej, który został przypisany w konstruktorze lub za pomocą metody setKey(). Tego typu klucz jest specyficzny dla aplikacji zaimplementowanych w Qt i rożni się od natywnego klucza nativeKey(), wykorzystywanego przez system. Metoda zwraca pusty łańcuch znaków w przypadku gdy klucz nie został ustawiony lub użyto natywnego klucza.
Bool	lock()	Metoda wykorzystywana do synchronizacji dostępu do segmentu pamięci współdzielonej, korzystająca z semaforów. Zwraca wartość true, gdy udało się zablokować segmentu pamięci na wyłączność obecnego procesu. W przypadku gdy inny proces blokuje dostęp do segmentu, wywołanie powoduje zablokowanie bieżącego procesu do momentu zwolnienia blokady przez proces blokujący i nałożenia jej przez proces bieżący. Wartość false jest zwracana w przypadku stosowania natywnego klucza setNativeKey() lub jeśli nie udało się użyć semafora z powodu błędu systemowego.
QString	nativeKey() const	Zwraca natywny, specyficzny dla platformy klucz, ustawiony za pomocą metody setNativeKey(). Natywny klucz może być wykorzystywany do dowiązania do segmentu pamięci współdzielonej, zarówno przez aplikacje zaimplementowane w Qt jak i przez aplikacje wykonane w innych technologiach. W przypadku gdy klucz został ustawiony za pomocą setKey(), metoda zwraca przekonwertowany klucz do postaci klucza specyficznego dla platformy.
Void	setKey(const QString &key)	Pozwala na ustawienie klucza do pamięci współdzielonej, specyficznego dla aplikacji zaimplementowanych w Qt. Tego rodzaju klucz jest niezależny od platformy. W przypadku gdy następuje zmiana klucza, a obiekt jest dowiązany do segmentu pamięci współdzielonej, następuje jego odwiązanie. Aby ponownie dowiązać obiekt należy skorzystać z metody attach().
Void	setNativeKey(const QString &key)	Ustawia klucz do pamięci współdzielonej, specyficzny dla danej platformy. W przypadku gdy nastąpi zmiana klucza, a obiekt jest dowiązany do segmentu pamięci współdzielonej obiekt zostanie automatycznie odwiązany. Aby ponownie dowiązać obiekt należy skorzystać z metody attach(). Użycie tego rodzaju klucza sprawia, że aplikacja nie będzie wieloplatformowa.
Int	size() const	Metoda zwraca wielkość (w bajtach) segmentu pamięci współdzielonej, do którego jest dowiązany obiekt, w przypadku gdy obiekt nie jest dowiązany jest zwracana wartość 0.
Bool	unlock()	Jeżeli blokada na segment pamięci współdzielonej została założona przez ten proces, następuję odblokowanie i zwrócenie wartości true. Jeżeli blokada nie została założona lub jeśli blokadę założył inny proces, metoda zwraca wartość false.

Listing 1. Plik nagłówkowy klasy Producent

```
1 #include <QObject>
2 class QSharedMemory;
3 class Producent: public QObject
4 {
5     Q_OBJECT
6 public:
7     explicit Producent (QObject *rodzic = 0);
8
9 public slots:
10     void start();
11
12 private:
13     static const QString sc_klucz;
14     static const int sc_maxIloscProduktow;
15     static const int sc_maxRozmiarPamieci;
16     QSharedMemory *m_pamiecDzielona;
17     int m_wyprodukowano = 0;
18     QByteArray m_dane;
19     int losujLiczbe(int min, int max);
20     bool obsluzBlad(bool blad, const QString &tresc);
21 };
```

Listing 2 przedstawia plik źródłowy klasy Producent. W linijkach 8-11 znajdują się definicje stałych, w konstruktorze klasy jest tworzony obiekt klasy QSharedMemory i ustawiany klucz na wartość "producent-konsument-klucz". Na początku metody start() jest tworzony segment pamięci o wskazanej wielkości przy pomocy metody create(), jeżeli segment pamięci został utworzony poprawnie, zostaje wypisany na wyjście klucz, natomiast w przypadku niepowodzenia komunikat o błędzie. Niepowodzenie również skutkuje zakończeniem aplikacji (linia 20). W linijkach 31-32 jest pobierany czas, który zostaje wykorzystany, aby zainicjować generator liczb pseudolosowych wykorzystywany przez metodę losujLiczbe(). Linijki 35-71 zawierają główną pętle aplikacji, na początku pętli następuje próba zablokowania segmentu pamięci na wyłączność danego procesu (linia 36), w przypadku gdy inny proces blokuje dostęp do segmentu, wywołanie powoduje zablokowanie bieżącego procesu do momentu zwolnienia blokady przez proces blokujący i nałożenia jej przez proces bieżący. Po nałożeniu blokady następuje pobranie zawartości pamięci współdzielonej i sprawdzenie jej rozmiaru. Jeśli pamięć jest pełna blokada jest opuszczana i program przechodzi do ponownego rozpoczęcia pętli, co daje czas innemu procesowi (konsumentowi) na dostęp do pamięci. W przypadku gdy rozmiar danych jest mniejszy od maksymalnie dopuszczalnego, następuje przystąpienie do wyprodukowania produktu i umieszczenia go w pamięci. Jest to realizowane w linijkach 51-56, na początku aplikacja losuje liczbę i dopisuję ją na końcu bufora, następnie zawartość bufora zostaje skopiowana do segmentu pamięci współdzielonej oraz następuje opuszczenie blokady. Linijki 58-59 odpowiadają za aktualizacje stanu produkcji - inkrementacje ilości wyprodukowanych produktów oraz dodanie kolejnego produktu (liczby) do tablicy bajtów, gdzie znajdują się kolejno wyprodukowane produkty. Linijki 61-70 są odpowiedzialne za przerwanie wykonywania programu gdy zostanie osiągnięty limit produkcji, w tym przypadku zostają wypisane na wyjście: informacja o ilości wyprodukowanych produktów, funkcja skrótu wyliczona dla zgromadzonych danych (MD5) oraz komunikat o zakończeniu.

Listing 2. Plik źródłowy klasy Producent

```
1 #include "producent.h"
2 #include <QCoreApplication>
3 #include <QSharedMemory>
4 #include <QDebug>
5 #include <QTime>
6 #include <QCryptographicHash>
7
8 const QString Producent::sc_klucz = "producent-
9 konsument-klucz";
10 const int Producent::sc_maxIloscProduktow = 200000;
11 const int Producent::sc_maxRozmiarPamieci = 4096;
12
13 Producent::Producent(QObject *rodzic)
14   : QObject(rodzic),
15    m_pamiecDzielona{new QSharedMemory(sc_klucz,
16 this)} {}
17
18 void Producent::start()
19 {
20   if (obsluzBlad(!m_pamiecDzielona-
21 >create(sc_maxRozmiarPamieci),
22                  "Utworzenie segmentu pamieci
23 wspoldzielonej nie bylo mozliwe:")) {
24       return;
25    } else {
26       qDebug () << "Utworzono segment pamieci
27 wspoldzielonej o kluczu:"
28                 << m_pamiecDzielona->key();
29   }
30    QByteArray bufor;
31    QTime czas = QTime::currentTime();
32    qsrand((uint)czas.msec());
33    int produkt = 5;
34
35   while (true) {
36       if (obsluzBlad(!m_pamiecDzielona->lock(),
37 "Zakladanie blokady nie powiodlo sie:"))
38           return;
39
40       char *zawartosc = (char*)m_pamiecDzielona-
41 >data();
42       bufor = zawartosc;
43       if (bufor.size() + 1 >= sc_maxRozmiarPamieci) {
44           if (obsluzBlad(! m_pamiecDzielona->unlock(),
45 "Zdejmowanie blokady nie powiodlo sie:"))
46               return;
47
48           continue;
49       }
50
51       produkt = losujLiczbe(1, 9);
52       bufor.append(QByteArray::number(produkt));
53       strcpy(zawartosc, bufor.constData());
54
55       if (obsluzBlad(! m_pamiecDzielona->unlock(),
56 "Zdejmowanie blokady nie powiodlo sie:"))
57           return;
58       ++m_wyprodukowano;
59      m_dane.append(QByteArray::number(produkt));
60
61       if (m_wyprodukowano == sc_maxIloscProduktow)
62
63           qDebug () << "Wyprodukowano" <<
64 m_wyprodukowano << "produktow.";
65           qDebug () << "MD5:" <<
66 QCryptographicHash::hash(m_dane,
67 QCryptographicHash::Md5).toHex();
68           qDebug () << "Zakonczono.";
```

```
69            QCoreApplication::exit(0);
70            return;
71        }
72    }
73 }
74
75 int Producent::losujLiczbe(int min, int max)
76 {
77     return qrand() % ((max + 1) - min) + min;
78 }
```

Interfejs klasy Konsument, jest bardzo podobny do interfejsu klasy Producent. W klasie została zadeklarowana metoda start(), która rozpoczyna pobieranie danych oraz metoda obsluzBlad(), do prostej obsługi błędów. W sekcji prywatnej umieszczono deklaracje typów: sc_klucz jest to zmienna przechowująca klucz do pamięci współdzielonej w postaci łańcucha znaków, sc_maxIloscProduktow określa ilość produktów jakie mają zostać pobrane, m_pamiecDzielona to wskaźnik na obiekt wystawiający interfejs do operowania na segmentach pamięci współdzielonej, zmienna m_pobrano jest wykorzystywana do zliczania pobranych produktów, natomiast obiekt m_dane gromadzi pobrane produkty (liczby) w postaci tablicy bajtów.

Listing 3 przedstawia plik źródłowy klasy Konsument. Stałe oraz klucz do pamięci współdzielonej są takie same jak w klasie Producent. W metodzie start() jest podejmowana próba dowiązania do segmentu pamięci współdzielonej przy pomocy wywołania metody attach(), jeżeli segment pamięci o danym kluczu istnieje i udało się dowiązać zostaje wypisany na wyjście klucz, natomiast w przypadku niepowodzenia nastąpi wypisanie komunikatu o błędzie i zakończenie wykonywania aplikacji (linia 18). Linijki 27-58 zawierają główną pętle aplikacji, podobnie jak w klasie Producent, na początku pętli następuje próba zablokowania segmentu pamięci na wyłączność danego procesu, metodą lock() (linia 28), w przypadku gdy inny proces blokuje dostęp do pamięci, wywołanie powoduje zablokowanie bieżącego procesu do momentu zwolnienia blokady przez proces blokujący i nałożenia jej przez proces bieżący. Po nałożeniu blokady, następuje pobranie zawartości pamięci współdzielonej do bufora i sprawdzenie jej rozmiaru. Jeśli pamięć jest pusta, blokada jest opuszczana i program przechodzi do ponownego rozpoczęcia pętli, co daje czas procesowi producenta na umieszczenie w magazynie nowych produktów. W przypadku gdy rozmiar danych jest większy od 0, pamięć współdzielona zostaje wyczyszczona, po wyczyszczeniu następuje opuszczenie blokady (linijki 41-44). Linijki 45-46 odpowiadają za aktualizacje stanu przetwarzania - zapisanie danych z bufora do tablicy bajtów, która przechowuje pobrane produkty oraz zwiększenie liczby pobranych elementów o wielkość bufora. Linijki 48-57 są odpowiedzialne za zakończenie programu gdy zostanie osiągnięty limit pobranych danych, w tym przypadku zostają wypisane na wyjście: informacja o ilości pobranych produktów, funkcja skrótu wyliczona dla przechowywanych danych (MD5) oraz komunikat o zakończeniu.

Listing 3. Plik źródłowy klasy Konsument

```
1 #include "konsument.h"
2 #include <QCoreApplication>
3 #include <QSharedMemory>
4 #include <QDebug>
5 #include <QCryptographicHash>
6
7 const QString Konsument::sc_klucz = "producent-
8 konsument-klucz";
9 const int Konsument::sc_maxIloscProduktow = 200000;
10
11 Konsument::Konsument(QObject *rodzic)
12   : QObject(rodzic),
13    m_pamiecDzielona{new QSharedMemory(sc_klucz,
14 rodzic)} {}
15
16 void Konsument::start() {
17    if (obsluzBlad(!m_pamiecDzielona->attach(),
18                "Dowiazanie do segmentu pamieci
19 wspoldzielonej nie bylo mozliwe:")) {
20        return;
21    } else {
22        qDebug () << "Dowiazano segment pamieci
23                wspoldzielonej o kluczu:"
24                << m_pamiecDzielona->key();
25    }
26    QByteArray bufor;
27   while (true) {
28        if (obsluzBlad(!m_pamiecDzielona->lock(),
29 "Zakladanie blokady nie powiodlo sie:"))
30            return;
31        char *zawartosc = (char*)m_pamiecDzielona-
32 >data();
33        bufor = zawartosc;
34        if (!bufor.size()) {
35            if (obsluzBlad(! m_pamiecDzielona->unlock(),
36                "Zdejmowanie blokady nie powiodlo sie:"))
37                return;
38            continue;
39        }
40
41        zawartosc[0] = 0;
42        if (obsluzBlad(! m_pamiecDzielona->unlock(),
43            "Zdejmowanie blokady nie powiodlo sie:"))
44            return;
45        m_dane.append(bufor);
46        m_pobrano += bufor.size();
47
48        if (m_pobrano == sc_maxIloscProduktow) {
49            qDebug () << "Pobrano" << m_pobrano <<
50                        "produktow.";
51            qDebug () << "MD5:" <<
52                QCryptographicHash::hash(m_dane,
53                QCryptographicHash::Md5).toHex();
54            qDebug () << "Zakonczono.";
55            QCoreApplication::exit(0);
56            return;
57        }
58    }
59 }
```

4. Wnioski

Klasa QSharedMemory udostępnia rozbudowany interfejs do pracy z pamięcią współdzieloną, pozwala zarówno na łatwe utworzenie obszaru pamięci współdzielonej o dowolnym rozmiarze, jak i dowiązanie do niego, a także dba o jego usunięcie w momencie zakończenia pracy ostatniego dowiązanego procesu. Do identyfikacji segmentu pamięci można wykorzystać dwa rodzaje kluczy: specyficzny dla Qt oraz specyficzny dla

danej platformy. Takie rozwiązanie pozwala na tworzenie obszaru pamięci współdzielonej przez aplikację zaprogramowaną w Qt i korzystanie z tego obszaru przez aplikacje wykonane w innych technologiach. Interfejs klasy pozwala również na łatwe śledzenie błędów, błędy mogą być pobierane w postaci typu wyliczeniowego lub w postaci łańcucha znaków - krótkiego opisu słownego. Programista może określić, w łatwy sposób, w jakim trybie będzie realizowany dostęp do pamięci - proces może otrzymać prawo do zapisu i odczytu lub wyłącznie do odczytu. Operacja zapisu lub odczytu jest realizowana w stylu języka C, interfejs klasy nie pozwala na łatwy zapis oraz sprawdzenie wielkości zapisanych danych, a więc programista korzystający z tej metody musi znać podstawy języka C. Klasa QSharedMemory posiada wbudowany mechanizm synchronizacji dostępu do pamięci, który okazał sie bardzo prosty w użyciu, podczas implementacji przykładowych aplikacji. Spójność danych została sprawdzona za pomocą funkcji skrótu (MD5), wyliczonej na wygenerowanych i pobranych danych. Wyniki pracy programów zostały przedstawiona na rysunku 1 i 2. Zarówno w aplikacji producenta jak i aplikacji konsumenta otrzymano taki sam skrót.

```
Utworzono segment pamieci wspoldzielonej o kluczu: "producent-konsument-klucz"
Wyprodukowano 200000 produktow.
MD5: "0cd02cf2a1f2c22bccd8229fbc3537f1"
Zakonczono.
```

Rys. 1. Wynik pracy aplikacji Producent.

```
Dowiazano segment pamieci wspoldzielonej o kluczu: "producent-konsument-klucz"
Pobrano 200000 produktow.
MD5: "0cd02cf2a1f2c22bccd8229fbc3537f1"
Zakonczono.
```

Rys. 2. Wynik pracy aplikacji Konsument.

Literatura

[1] J. S. Gray, Komunikacja między procesami w Unixie. ReadMe, Warszawa 1998

[2] John Fusco, The Linux Programmer's Toolbox, Pearson Education, 2007

[3] http://doc.qt.io/qt-5/ipc.html [24.05.2017]

[4] http://wiki.qt.io/Qt_History [24.05.2017]

[5] Symeon Huang, Qt 5 Blueprints, Pack Publishing, 2015

[6] Guillaume Lazar, Robin Penea, Mastering Qt 5, Packt Publishing , 2016

[7] http://doc.qt.io/qt-5/licensing.html [24.05.2017]

[8] A. Ezust, P. Ezust, Qt. Wprowadzenie do wzorców projektowych. Wydanie II, Helion, 2014

[9] http://doc.qt.io/qt-5/topics-app-development.html [24.05.2017]

[10] http://doc.qt.io/qt-5/metaobjects.html [24.05.2017]

[11] http://doc.qt.io/qt-5/signalsandslots.html [24.05.2017]

[12] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: Wzorce projektowe. Elementy oprogramowania obiektowego wielokrotnego użytku. Helion, 2010

[13] http://doc.qt.io/qt-5/objecttrees.html [24.05.2017]

[14] http://doc.qt.io/qt-5/qsharedmemory.html [24.05.2017]